Incremental Modernization of Legacy Systems

Santiago Comella-Dorda
Grace A. Lewis
Pat Place
Dan Plakosh
Robert C. Seacord

July 2001

COTS-Based Systems

Technical Note
CMU/SEI-2001-TN-006

Contents

List of Figures

Abstract

This report shows an objective technique for developing an incremental code-migration strategy for large legacy Common Business-Oriented Language (COBOL) systems. Specifically, it describes a case study that involves the modernization of a large Supply System (SS). The system consists of approximately 2 million lines of COBOL code operating in a mainframe environment.

The SEI developed the System Analysis and Migration (SAM) tool to generate a code migration strategy based upon legacy system analysis data. SAM considers a set of factors that includes minimizing scaffolding code (code that is discarded before the completion of the project), balancing iterations, and grouping related functionality.

1 Background

Modernization of legacy information systems are often large, multiyear projects that pose significant risks. Information systems are critical to companies, and making a single deployment of the modernized version is too risky to be admissible. Additionally, a modernization effort of a large system requires a significant investment in terms of money and time; projects of this magnitude are strongly pressured to demonstrate early benefits. Figure 1 illustrates a typical example in which the modernized version of the system is developed and deployed incrementally over a six-year period.

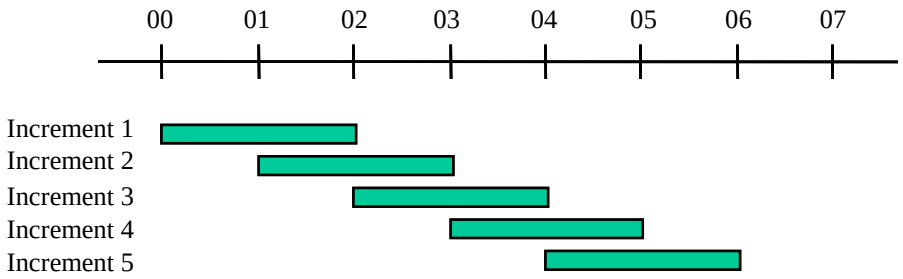

Figure 1: Development Increments

Incremental modernization of a legacy system is illustrated in Figure 2. Initially, the legacy system consists completely of legacy code. At the completion of each increment, the percentage of legacy code decreases while the percentage of modernized code increases. Eventually, the system is completely modernized.

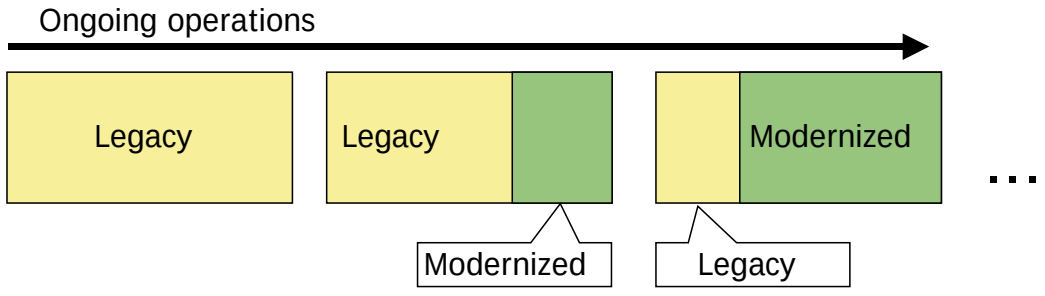

Figure 2: Legacy System Modernization

Since modernized components are being deployed prior to the completion of the entire system, it is necessary to combine elements from the legacy system with the modernized components to maintain the existing functionality during the development period. Adapters and other wrapping techniques may be needed to provide a communication mechanism between the legacy system and the modernized system, when dependencies exist.

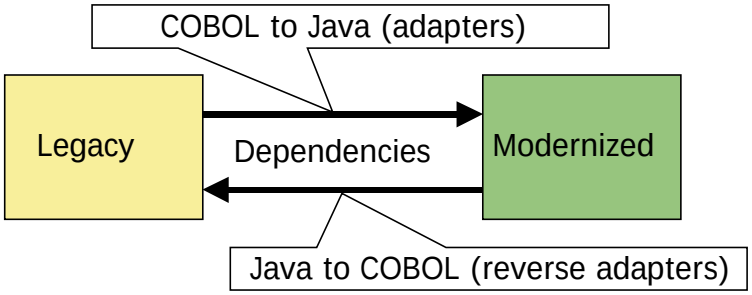

Figure 3: Interoperation of the Modernized and the Legacy Systems

An incremental modernization effort strives to keep the system fully operational at all times while reducing the amount of rework and technical risk during modernization. In order to balance these conflicting requirements, a modernization effort needs to be carefully planned. Planning a modernization effort does not only involve creating a budget and setting up a few milestones. A modernization plan must also contain the order in which the functionality is going to be modernized, along with information describing the scaffolding code that must be created to keep the system operational at all times.

The technical aspects of the modernization plan are what we call the modernization strategy. In this report we describe a case study that involves the creation of such strategy for the modernization of a large Supply System (SS). The SS consists of approximately two million lines of COBOL code running on a mainframe. The overall architecture of the system has remained largely unchanged over 30 years, resulting in a system that is extremely brittle and difficult to maintain. (A relevant description of an information system life cycle is provided by Comella-Dorda, et. al.[Comella 00]).

This report is structured as follows: Section 2 describes the factors and tradeoffs that must be considered in the migration strategy. Section 3 presents the detailed processes that were followed to generate the code migration strategy. Section 4 describes the Unified Modeling Language (UML) notation that was used to document the migration strategy. In the final section, conclusions are presented.

2 Modernization Strategy

As we implied previously, the single most import factor in developing the code migration strategy is to keep the system fully functional at all times. Since some subset of functionality is completed at the end of each development and deployment increment, it is necessary to maintain the use of the legacy system to provide functionality not already modernized. In addition to meeting this requirement, certain goals and objectives are inherent in defining a componentization strategy. These include

- Minimize development and deployment costs. Fielding modernized components alongside legacy code requires the development of adapters, bridges, and other scaffolding code that will be discarded after the final increment. While necessary, scaffolding code represents an added expense, as this code must be designed, developed, tested, and maintained during the development period. Minimizing the development of scaffolding code is one way to minimize overall development costs.
- Support an aggressive yet predictable schedule. The componentization strategy should seek to minimize the time required to develop and deploy the modernized system. Additionally, the approach should allow the system to be developed on a predictable schedule.
- Maintain quality of interim and final products. There are two issues regarding quality. One is the quality of the final, end-state system, once the modernization effort has been completed. The final system should be easy to maintain and implemented around technologies that are not already obsolete. The second issue is the interim quality of the system after each increment is deployed. Given the length of time required to modernize a system, there are many opportunities for the development effort to lose funding, be redirected, or take on a new focus. It is important that each fielded increment improve the overall quality of the system; there is always the possibility that each increment will be the last due to the normal uncertainties caused by changing business practices and requirements
- Minimize risk. Risks occur in many different forms, and some risk is acceptable if it is managed and mitigated properly. Due to the overall size and investment required to complete a system migration, it is important that overall risk be kept low. To this end, the componentization strategy should apply tried-and-proven techniques when possible, and lower-risk approaches when some risk is necessary to achieve overall system goals.
- Meet system performance expectations. The modernized system will replace an existing system, so users have expectations concerning performance. While modernization often includes hardware as well as software components, it is easy to diminish hardware performance gains with poorly designed software. The componentization strategy must ensure that user performance expectations are met or exceeded.
- Maintain complexity at a manageable level. The chaotic structure and size of most legacy information systems is a major complexity factor by itself. As a result, it is critical that the componentization strategy seek to minimize overall system and development complexity, so that the latter is kept at a manageable level. Managing the complexity of

the development approach may be the single largest factor that dictates the viability of the overall modernization effort.

2.1 System Architecture

In the modernization of any legacy system there are groups of stakeholders with varying opinions on how to proceed. It is important to develop consensus among these stakeholders before moving forward. As part of the case study we conducted a componentization workshop, in which the stakeholders agreed on the general approach to modernize the system. The group consensus included a decision to migrate the existing legacy code based on the existing structure of the legacy elements. The structure and dependencies of the legacy system are analyzed to determine the simplest way to extract functionality from the legacy system to be modernized. This approach reduced the overall complexity and risk in ensuring that the incrementally deployed systems retained the overall functionality of the legacy system.

A potential problem adopting this tactic is that the architecture of the modernized system may be constrained by the structure of the legacy system. In fact, as logical functionality chunks of the legacy system are extracted, the developer tends to replicate the same chunks in the modernized system. If this problem were not considered, we would end up with the same system running on a different platform.

In order to avoid this problem we need two things. First, a well-defined target architecture that has been developed taking into account, but not being constrained by, the structure of the legacy system. And second, a mechanism to decouple the functional partition of the legacy and modernized systems even if those systems share mutual dependencies. This mechanism has to make it possible to split program elements across components in the modern architecture. These components can be deployed while still incomplete—as long as the overall functionality of the system remains intact.

The target architecture in the case study is based on the OAGIS specification [OAG 99], which prescribes a loosely coupled architecture[1]. This architecture consists of business objects, which are software components that encapsulate the business logic of a single entity and data particular to that entity. Examples of OAGIS business objects include order management, accounts receivable, and general ledger.

The mechanism used to decouple the structure of both systems is based on adapters. Figure 4 illustrates code migration by program element sets and the use of adapters. A legacy program element (121) is scheduled for modernization. The functionality performed by this program element is re-implemented as part of the modernized architecture as shown on the right. However, program element 121 is still invoked by program element 345, and invokes

[1] Lewis, Grace & Comella-Dorda, Santiago. *Data Architecture Guide for the ILS-S System*, to be published in September 2001 by the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

program element 129, neither of which has been modernized in this example. In this case, it is necessary to develop a shell and adapter for the program element. The shell ensures that the external interfaces of program element are maintained. The adapter accepts requests from the 121 shell and invokes methods in the modernized components to implement this functionality. Results can then be returned to the 121 shell, which will use this data to satisfy its external requirements.

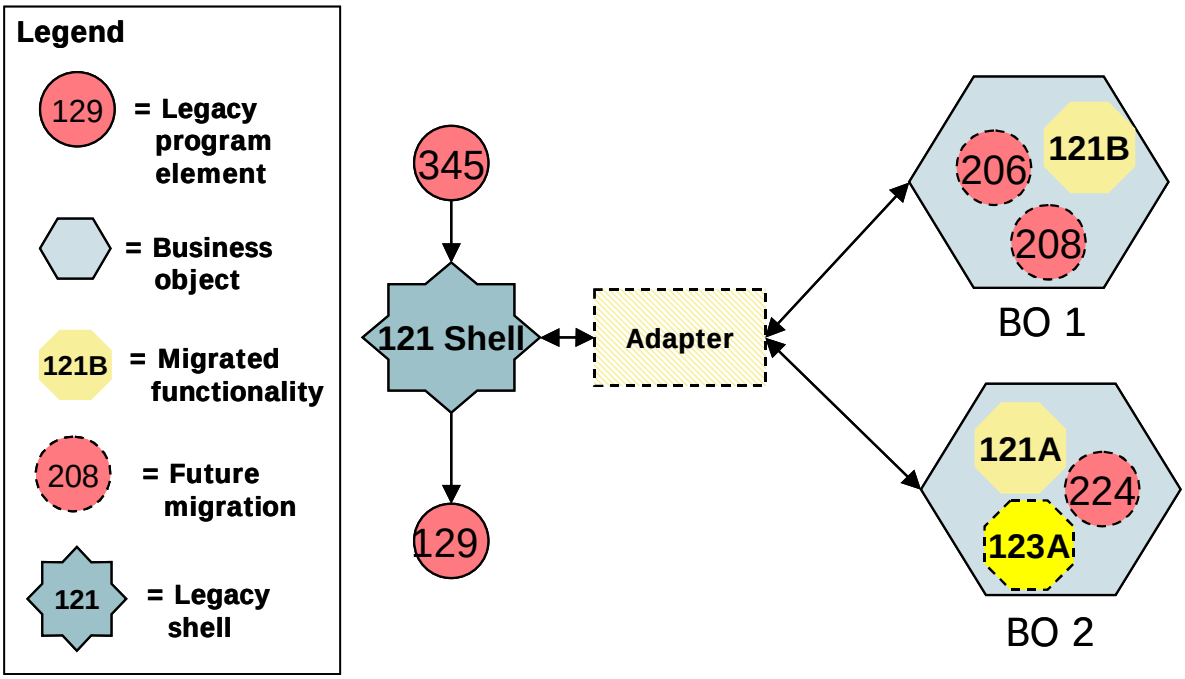

Figure 4: Sample Code Migration Using Program Element Sets

The development of the shell and adapter code is not trivial, however, and in a large system (with over 900 program elements in the case study) will result in the development of a significant amount of scaffolding code that will eventually be discarded. Reducing the amount of scaffolding code required should reduce overall development costs. In fact, reducing the number of adapters has been one of the main drivers when calculating the order in which to migrate program elements.

2.2 Migration Plan

One of the most important questions that must be answered in an incremental development and deployment process is "Which program elements will be modernized in each increment?" Unless some objective criteria are developed to answer this question, the migration plan is prone to be driven by forces external to the development process. Based on the overall modernization strategy goals described earlier in this report, the criteria selected for the case study were[2]

- Minimize the number of adapters.
- Migrate modules with related functionality at the same time.
- Map increments' size to the funding profile.

The first goal is to reduce the number of adapters. Reducing the number of adapters eliminates the need to develop scaffolding code that will eventually be discarded. Reducing the need for adapters should reduce the overall development costs for the system.

The second goal is to eliminate the need for developers to understand the entire system during the development of each increment. Unfortunately, optimizing the entire system to reduce the number of adapters may not be the best overall approach, as it does not take the commonality of program elements into account. If program elements for modernization are selected solely based on a global optimization to reduce adapters, program elements from widely varying functional areas of the system are likely to be grouped into a single iteration. This would cause developers to work in widely varying functional areas, making it difficult for them to comprehend and implement the system requirements.

The third goal maps the migration plan to available funding. In some situations funding may vary from year to year, and it may even be outside the control of the development team. As a result, increments must be mapped to available funding. Sometimes if this is impossible or technically undesirable, an increment can be adjusted to fit within the funding constraints by adjusting other cost drivers such as new functionality.

[2] The presentation order of these goals does not imply degree of importance.

3 Developing the Migration Strategy

Our starting point in analyzing the structure of the legacy system was to develop a call graph. Examining the calling structure of the legacy system helped us to identify program elements with minimal dependencies that could be migrated easily.

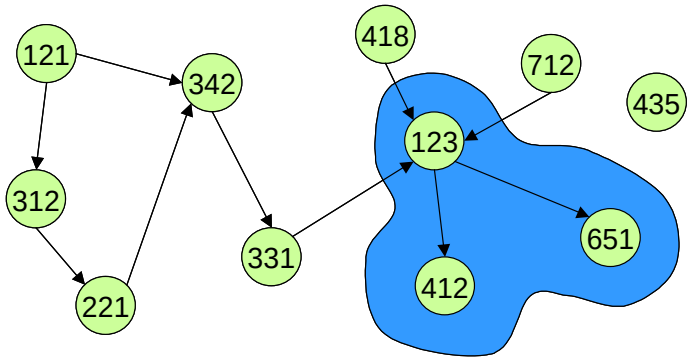

Figure 5: Call Graph

Figure 5 illustrates what we were hoping to achieve in this process. The circles in this figure represent program elements, while the arrows show call relationships. Evident in this figure are four different kinds of program elements:

1. *Root program elements* call other program elements, but are not called by any (e.g., program element 418).
2. *Leaf program elements* are called by program elements, but do not call any (e.g., program element 412).
3. *Node program elements* both call, and are called by, other program elements (e.g., program element 123).
4. *Isolated program elements* neither call nor are called by other program elements (e.g., program element 435).

Root program elements are typically programs that are invoked directly by the user or some external process (otherwise there would be no way to execute these programs). By themselves, these program elements may not be good candidates for modernization since they call other program elements. This means that the modernized Java components would have to call back to the COBOL system—something we were trying to avoid, as we didn't want the execution control going back and forth between the modernized and the legacy system.

Node program elements are even more difficult to migrate than root program elements. They share the difficulty of root program elements, but also require that adapters be created in the legacy code so that the remainder of the legacy system can continue to function in the same manner.

Independent of other issues, isolated program elements can be migrated easily. These elements could be used as "filler" in any given increment, since migrating them does not increase or decrease the number of adapters that need to be developed.

Leaf program elements are the best candidates for migration. They do not call back to the COBOL code, and although they require the development of adapters, it is possible to minimize the number of these adapters by migrating entire subtrees in a single iteration. The shaded area in Figure 5 shows where three program elements can be migrated requiring only a single adapter.

Initially, we assumed that we could use a commercial off-the-shelf COBOL analysis tool to generate a call graph for the legacy system. Our clients had already acquired a well-known analysis tool, which boasted significant functionality, the least of which was the ability to generate a call graph. Unfortunately, the system under consideration used a call mechanism apparently not supported by the tool. In particular, calls are made consistently by moving the name of the program to call into a variable, and then calling the value of the variable as shown below:

```
MOVE 'NGV129' PROGRAM-TO-CALL
CALL PROGRAM-TO-CALL
```

We did not find any commercial tool able to detect these calls using static analysis; they could be detected only through the dynamic, run-time analysis of the code or manual techniques. As no commercial tools were available to perform this level of dynamic analysis, we decided to apply a more manual approach using the analysis already available for the system. These analyses were basically text based call indexes, record references, and miscellaneous data about particular program elements.

3.1 The Initial Plan

To generate call graphs from the available input, we decided to develop a tool that used the Rational Rose Extensibility Interface (REI) to create UML diagrams. We named this program "SAM" for System Analysis and Migration tool. SAM converts program elements listed in the Calls Index to classes, and calls to associations. Each association was then labeled with the call type (i.e., Perform, Call, Copy). There were a total of 900 distinct program element names and 10,629 calls. The resulting UML diagram is shown in Figure 6.

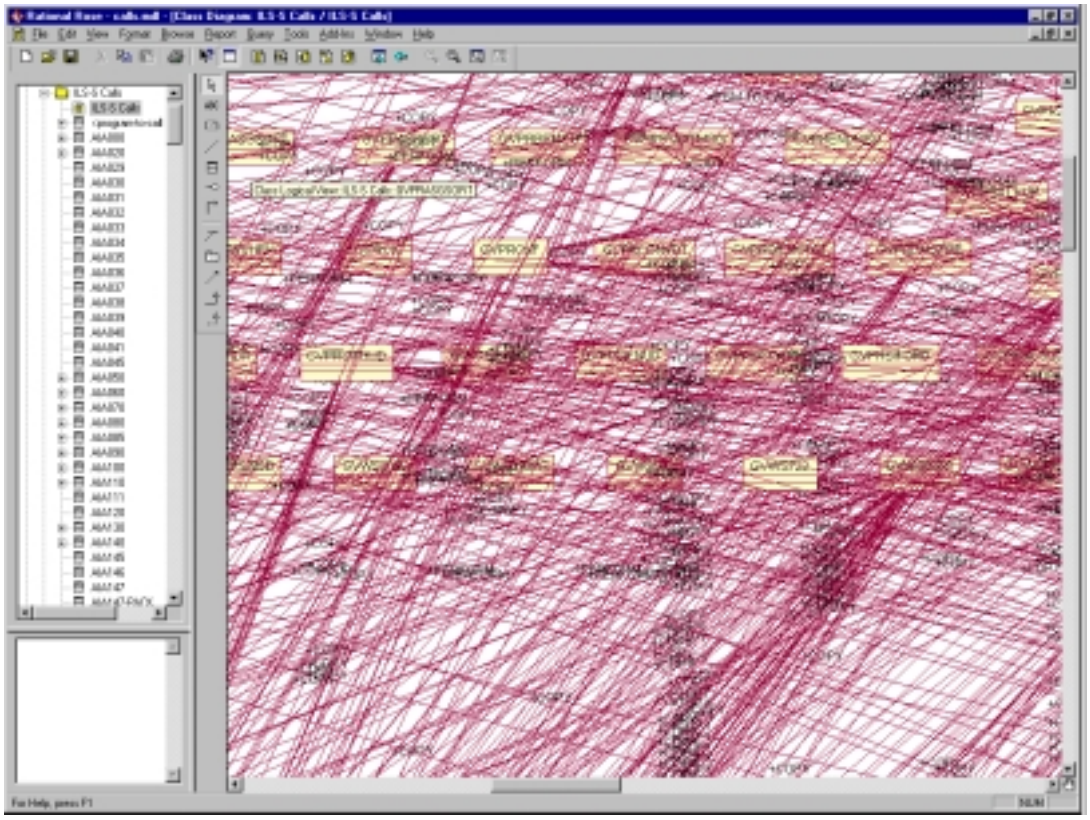

Figure 6: Initial Rational Rose Model

The initial Rational Rose model shown in Figure 6 is obviously too complex to be useful. Consequently, our process for generating this UML diagram required some refinement. We decided on several steps to simplify the model. First and foremost, we decided to create a separate diagram for each *root element*. Second, we eliminated self-referential elements. Third, we decided to reduce utility functions that typically fetch, delete, or update a database record to comments in caller program elements.

The result of this process was a little more comprehensible. A series of charts was generated and then analyzed. Our two primary findings from this portion of the study were in isolated program elements and root program elements.

There were a total of 96 root element diagrams in the system under consideration. Their complexity varies from 2 to over 100 elements; however most are in the range of 2-20 elements. Figure 7 shows an example of a relatively complex root element diagram. Our initial idea was to consider each of those diagrams as a modernization unit. All the program elements in a particular diagram will be modernized and deployed into the new system as a single unit.

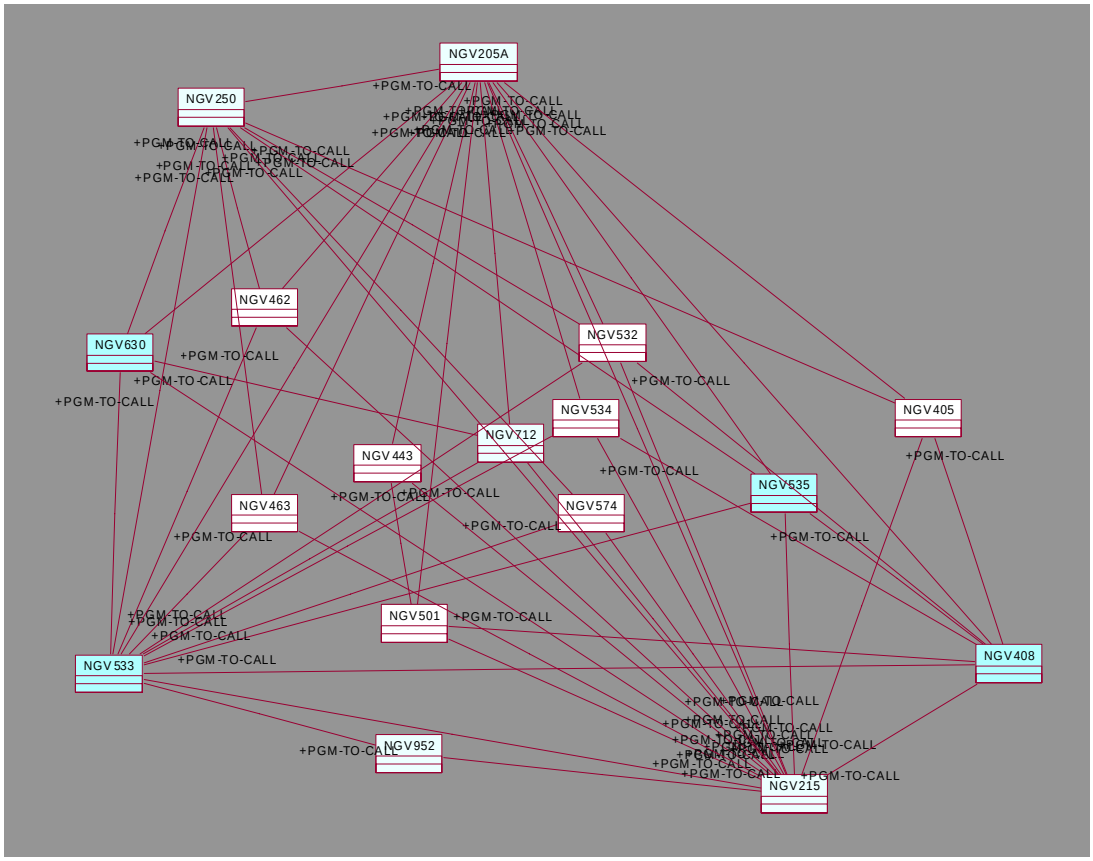

Figure 7: Root Element (NVG408) Diagram

The algorithm that we developed to generate the root element diagrams has some interesting features. To simplify the diagrams, subtrees for nodes within the diagram are expanded only once in the model. This missing complexity is captured in two ways: it is recorded in the documentation for the program element class and is displayed in the diagram using a shade of blue. Darkening shades of blue indicate increasing numbers of *reachable* elements, even of all those elements are not explicitly represented. Reachable elements are program elements that can be called, either directly or indirectly, by the program element.

The documentation for each program element class contains the number of diagrams in which the class is included (root elements are only included in one diagram), the number of times that the program element is called, and the number of program elements that are called. The number of modules in the tree is the same as the number of reachable elements. As stated earlier, calls to utility functions are reduced to comments and also included in the documentation for each program element class. Figure 8 shows a sample of the documentation provided for each program element class.

```
NUMBER OF DIAGRAMS: 1
TIMES CALLED: 0
TIMES CALLER: 4
```

```
NUMBER OF MODULES IN TREE: 14
NOTES:
accesses: GVPRGETPCT
…
accesses: GVPRCALLER
accesses: GVPRCNF
added to diagram: NGV227
expanded in diagram: NGV227
```

Figure 8: Program Element Class

Of the 900 overall program elements in the system, 248 are isolated program elements. These program elements can be ported easily with little impact (i.e., they do not require adapters). However, many of these program elements are probably reports, which are a special case, as they do not use component interfaces and are highly dependent on the database structure. As a result, it may make sense to identify program elements that are reports and defer their implementation to the final data-migration phase. Because these reports operate against the legacy database, they can remain unchanged up until that time.

While this analysis of root program elements clearly teaches us something about the structure of the legacy system, it is not clear that root elements form the best candidates for componentization. Our analysis revealed that root elements map directly to user-level transactions. Business objects, on the other hand, are often built as augmented encapsulations of data entities. Consequently, components in the modernized system are more likely going to correspond to the original data entities than to transactions. If we follow the root approach, every diagram represented in the UML models would map to a number of business objects in the modernized system. This would create an unwanted complexity, as multiple business objects in the modernized system would be open for changes at the same time.

3.2 The Revised Plan

As we had concerns with building a code-migration plan around root program elements, we conceived a new approach of building the plan around database records. This approach consisted of the following steps:

1. Arrange data records into as many sets as the number of increments (five in the case study). Logically related groups of records are grouped together in an attempt to achieve our goal of modernizing related functionality in each increment.
2. For each data record set
    a. Group together the program elements that reference or depend upon the database records on the set.
    b. Identify program elements outside the group that invoke programs inside the group (these potentially require adapters).

3. For the system, determine the modernization order for the groups created in Step 2 that minimizes the number of adapters that must be built and balances the size of iterations.

We further reduced the complexity of the system and improved the quality of the migration plan by eliminating database records and program elements that were likely to be eliminated in the modernized system. For example, we identified database records that were used to maintain global constants or information about the system (that is, the number of user terminals), since this information would be managed differently in the modernized system.

Once the database records have been grouped, they are run through the tool and the results are analyzed. If these results are acceptable, the plan can then be manually tweaked and executed. If the results are not acceptable, the database records can be reorganized and run through the tool again. Figure 9 shows an overall process diagram for this revised process.

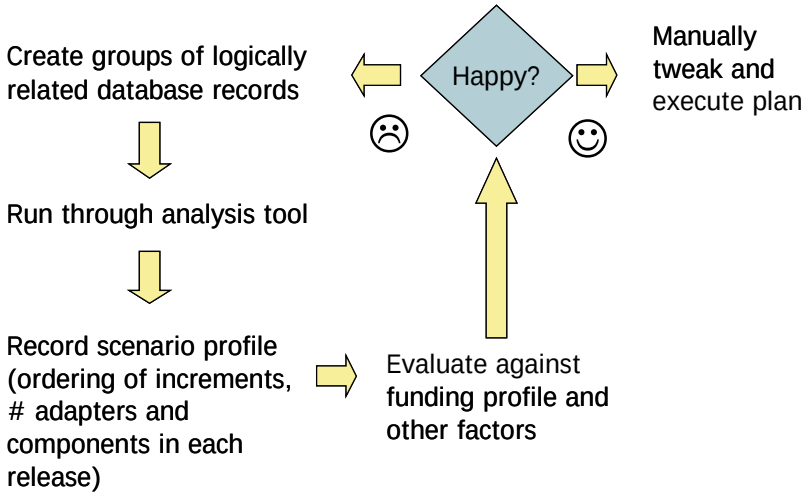

Figure 9: Process Overview

4 Describing the Migration Strategy Using UML

The results of the analysis described in the previous section are captured in a collection of Rational Rose UML models. A separate model is generated for each iteration. Each iteration contains a class diagram for each database record in the set.[3] Figure 10 shows a diagram for the 53 database record. The database record is displayed using a special icon. Program elements shown in white must be migrated as part of this increment. Program elements in green have already been migrated, either in a previous increment or earlier in this increment.

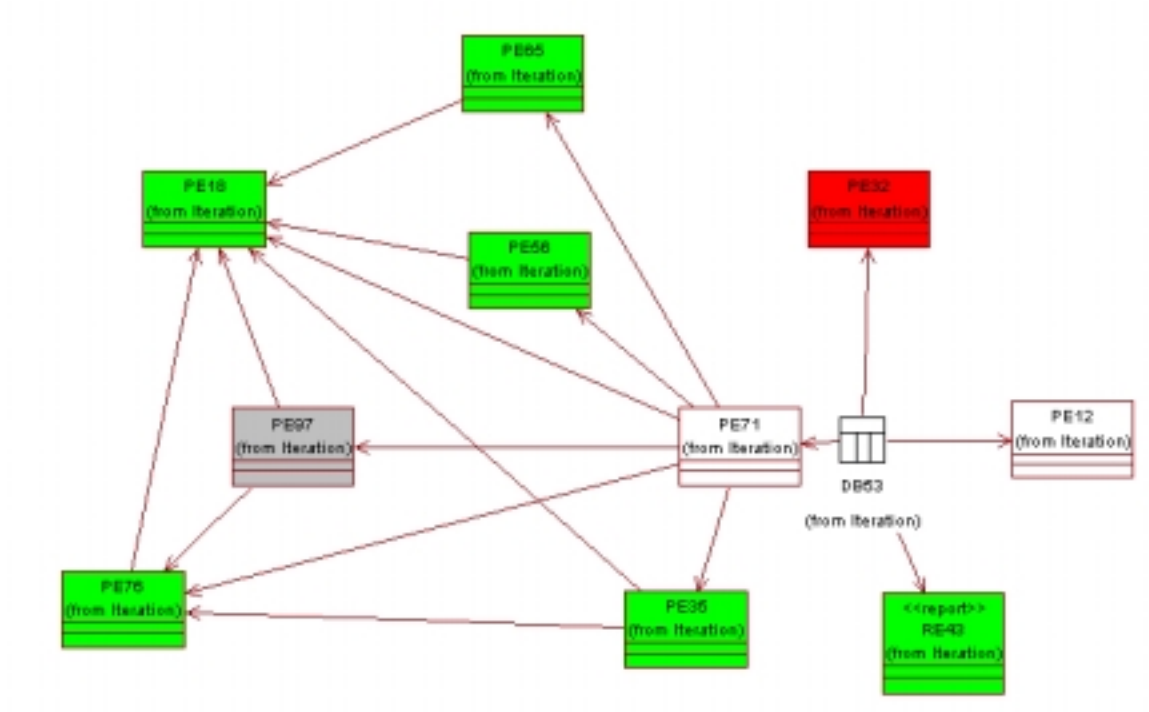

Figure 10: Database Element Set

Program elements in gray are elements that will be deleted. They are being included in the diagrams until they are actually removed. Their LOC is not being counted in any calculations, and their call graphs are not expanded. For example, if program element 1 calls program element 2 (which is deleted) and program element 2 calls program element 3, only program element 1 and program element 2 are represented, and program element 2 is represented in gray. Deleted program elements do not require adapters. If program element 1 is called by program element 2 that has not yet been ported but is marked as "deleted," program element 1 does not require an adapter.

[3] Database records without associated program elements are not included.

Program elements scheduled to be migrated that require the use of an adapter are shown in red. Adapters are required where program elements, not already ported, need to invoke a program element that is being migrated in the current increment.

Figure 11 shows a sample of the information captured for each iteration in the documentation for the Iteration package created in Rational Rose. This data includes the number of modules ported in the increment; the number of adapters required; the number of lines of code ported in this increment, both as an absolute value and as a percentage of the overall system; and the number of lines of executable code ported in this increment, both as an absolute value and as a percentage of the overall system. Following this header information is a description of the percentage of each business object[4] completed at the end of the iteration. There is one line for each business object. The number of lines of ported code associated with each business object is given, again as an absolute and as a percentage of the total lines of code in that business object. These calculations are performed using both the total lines of code (LOC) and executable LOC counts.

```
NUMBER OF MODULES PORTED: 37 NUMBER OF ADAPTORS REQUIRED: 8

 LOCs PORTED : 92308 (total= 1209996 percentage= 7)
     EXECUTABLE LOCs PORTED: 77941  (total= 1016515
     percentage= 7)
 BO COMPLETION (this iteration)
     Orders  LOCs: 7897 (total= 110611 percentage= 7)
     EXECUTABLE LOCs: 6074 (total= 88170 percentage= 6)
     Inventory  LOCs: 3435 (total= 165278 percentage= 2)
     EXECUTABLE LOCs: 2937 (total= 138815 percentage= 2)
     WIP_Confirm  LOCs: 1146 (total= 53331 percentage= 2)
     EXECUTABLE LOCs: 952 (total= 44859 percentage= 2)
     Requisition  LOCs: 1193 (total= 80729 percentage= 1)
     EXECUTABLE LOCs: 991 (total= 65271 percentage= 1)
     …
     Other LOCs: 0 (total= 1309 percentage= 0)
     EXECUTABLE LOCs: 0 (total= 1289 percentage= 0)
```

Figure 11: Sample Iteration Output

4.1 Results

We used these analysis techniques to generate a series of alternative code-migration strategies. Each strategy is contained in a number of UML models: one for each increment and a “final increment” containing the flexible allocation (isolated program elements). The strategies can be characterized related to some parameters, including the Lines Of Code

[4] The specific business objects are defined in the system architecture based on the OAGIS standard.

(LOC) allocated to each iteration, the number of necessary adapters, and the completion rate for business objects. It is useful to generate a profile summarizing these parameters for each potential strategy. The customer can then select the profile that better fits its resource allocation, or request additional profiles if no one is suitable. As an example, this section contains one of those profiles.

Figure 12 shows the LOC allocation for each increment. Iteration 1 is intentionally small, as the main concern of the first increment should be to explore the technical risks associated with the project. It should be possible to routinely apply lessons learned in the first iteration to later iterations. There is, of course, some management risk in this approach, as there will be a disproportionate number of dollars to lines of code ported after the first increment. This is a condition that will have to be supported by management fortitude.

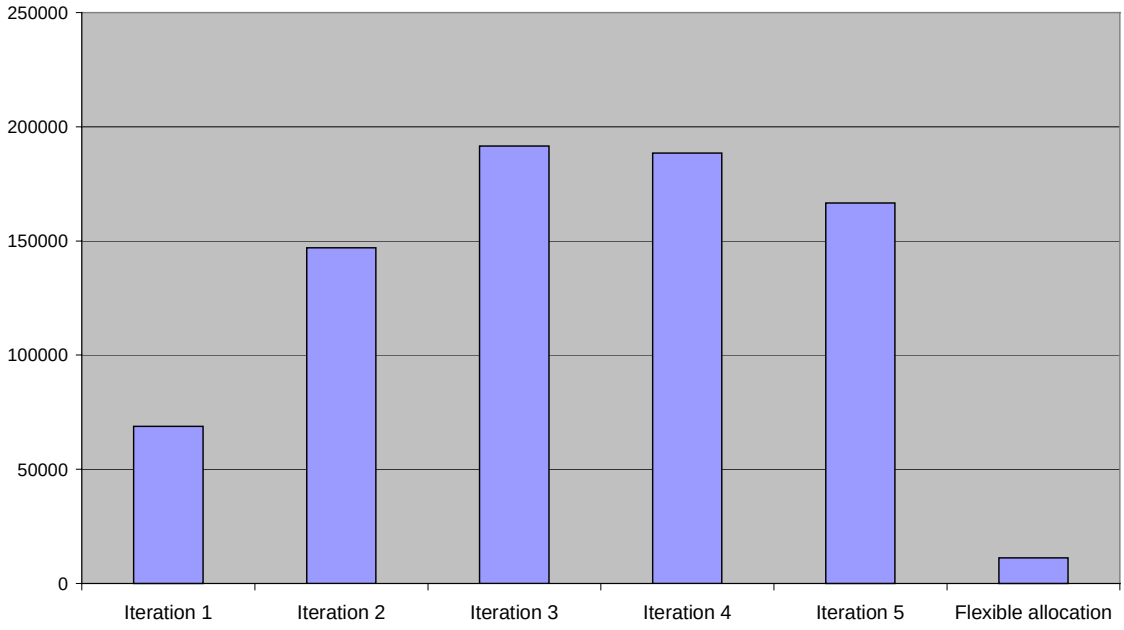

Figure 12: Effort per Iteration

Isolated program elements become part of the flexible allocation and can be assigned to any increment. This is convenient, as these isolated program elements can be used to increase the size of increments to more closely map to the funding profile. Ideally, the fixed allocation is below the projected effort for each increment. Program elements from the flexible allocation can then be added to each increment to bring them up to the projected effort for each increment.

Figure 13 shows the number of adapters (calls from the legacy code to the modernized code), inverse adapters (calls from the modernized code to the legacy code), and modules being ported and developed in each increment. The height of each bar shows the combined number of adapters and modules. Of course, no adapters are included in the flexible allocation.

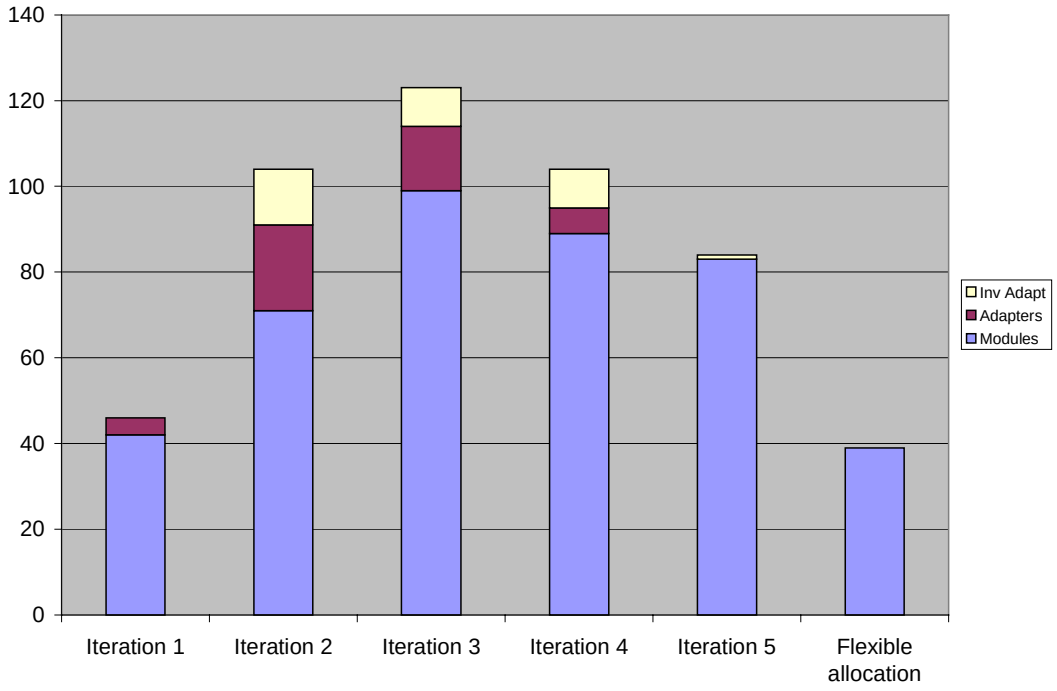

Figure 13: Adaptors vs. Modules

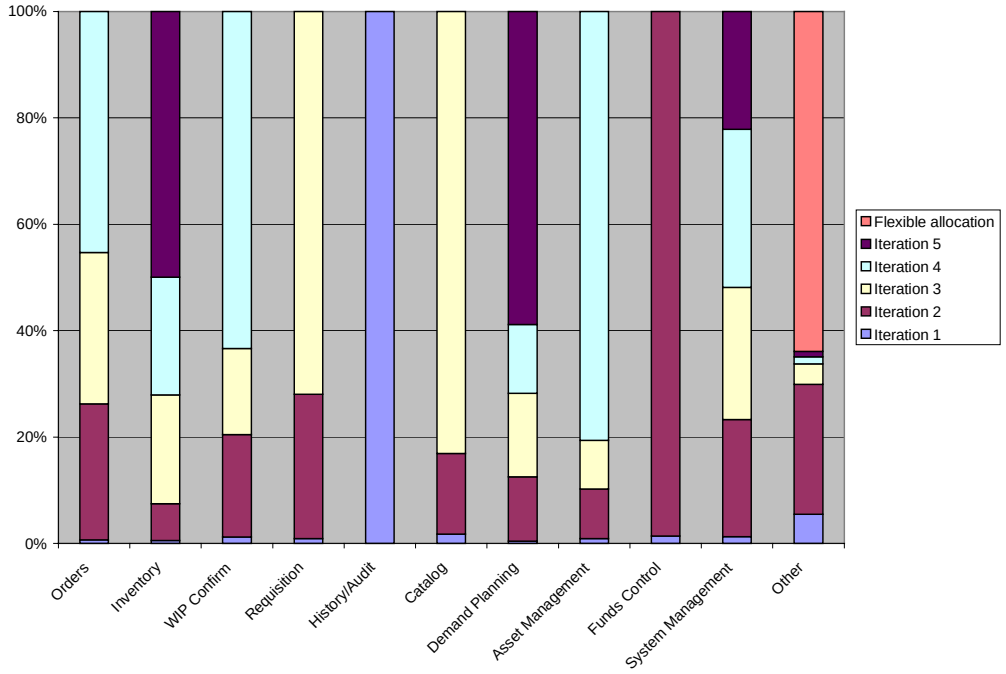

Figure 14: Business Object Completion per Iteration

Figure 14 shows the percentage of each business object completed after each iteration. Every iteration completes at least one business object. Completing business objects quickly is important in order to demonstrate early benefits and to finalize parts of the modernized system.

Figure 15 also shows the allocation of source lines of code (SLOC) to the business objects. This chart is useful in detecting business objects that have too much functionality (and should be decomposed) or that do not have enough (and should be merged). For example, it is not apparent that Match Doc should not even qualify as a business object. Other problems, like the large number of source lines allocated to system management must be considered.

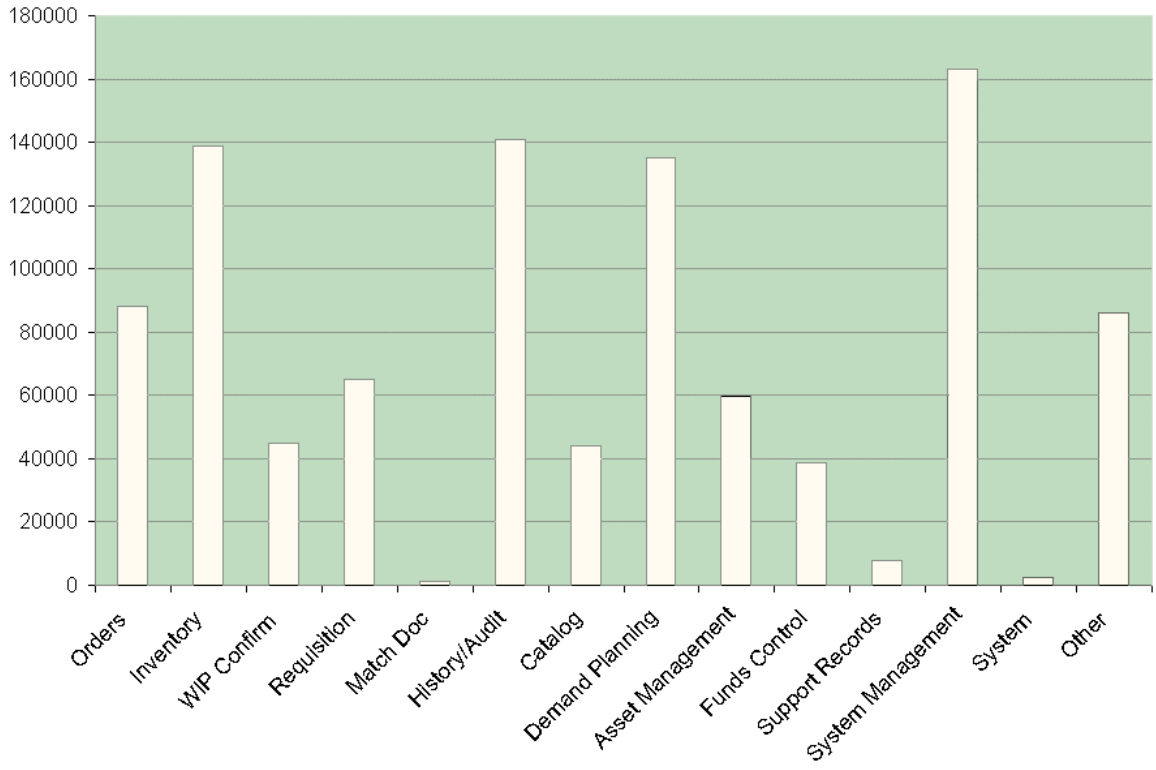

Figure 15: SLOC per Business Object

5 Conclusions

We have shown a technique for developing an incremental code migration strategy for large information systems. This technique provides a systematic and fact-based method that avoids the arbitrary, intuitive decision making too often found in software projects. The systematization of the technique has enabled us to partially automate the process through the creation of a tool that analyzes the legacy system and generates a code migration plan.

However, the strengths of this technique are also its weaknesses as the analysis tool necessarily lacks the insight and expertise only found in humans. The code migration plan created by the tool is generated automatically from a rigid set of predefined parameters, and the results should be treated accordingly. The plan must be tweaked by developers to accommodate particular concerns during the migration process. Nevertheless, the plan provides valuable information about the migration process.

We want to apply this analysis in other modernization efforts in order to fine-tune the techniques and the analysis tool. In particular, we are planning to improve the tool to take into account additional variables, including feedback from developers.

Bibliography

[Bobrowski 97] Bobrowski, Steve & Smith, Gordon. *Oracle8 Replication*. Oracle Corporation, 1997.

[Comella 00] Comella-Dorda, Santiago; Wallnau, Kurt; Seacord, Robert C.; & Robert, John. *Survey of Legacy System Modernization Approaches, A.* (CMU/SEI-2000-TN-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: URL< http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html>

[OAG 99] Open Applications Group. *Plug and Play Business Software Integration: The Compelling Value of the Open Applications Group*. Atlanta, GA: Open Applications Group, 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2001	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Incremental Modernization of Legacy Systems	F19628-00-C-0003

6. AUTHOR(S)

Santiago Comella-Dorda, Grace A. Lewis, Pat Place, Dan Plakosh, Robert C. Seacord

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2001-TN-006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report shows an objective technique for developing an incremental code-migration strategy for large legacy Common Business-Oriented Language (COBOL) systems. Specifically, it describes a case study that involves the modernization of a large Supply System (SS). The system consists of approximately 2 million lines of COBOL code operating in a mainframe environment.

The SEI developed the System Analysis and Migration (SAM) tool to generate a code migration strategy based upon legacy system analysis data. SAM considers a set of factors that includes minimizing scaffolding code (code that is discarded before the completion of the project), balancing iterations, and grouping related functionality.

14. SUBJECT TERMS	15. NUMBER OF PAGES
code migration, legacy, System Analysis and Migration, SAM, Retail Supply System modernization, RSS modernization	30

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL